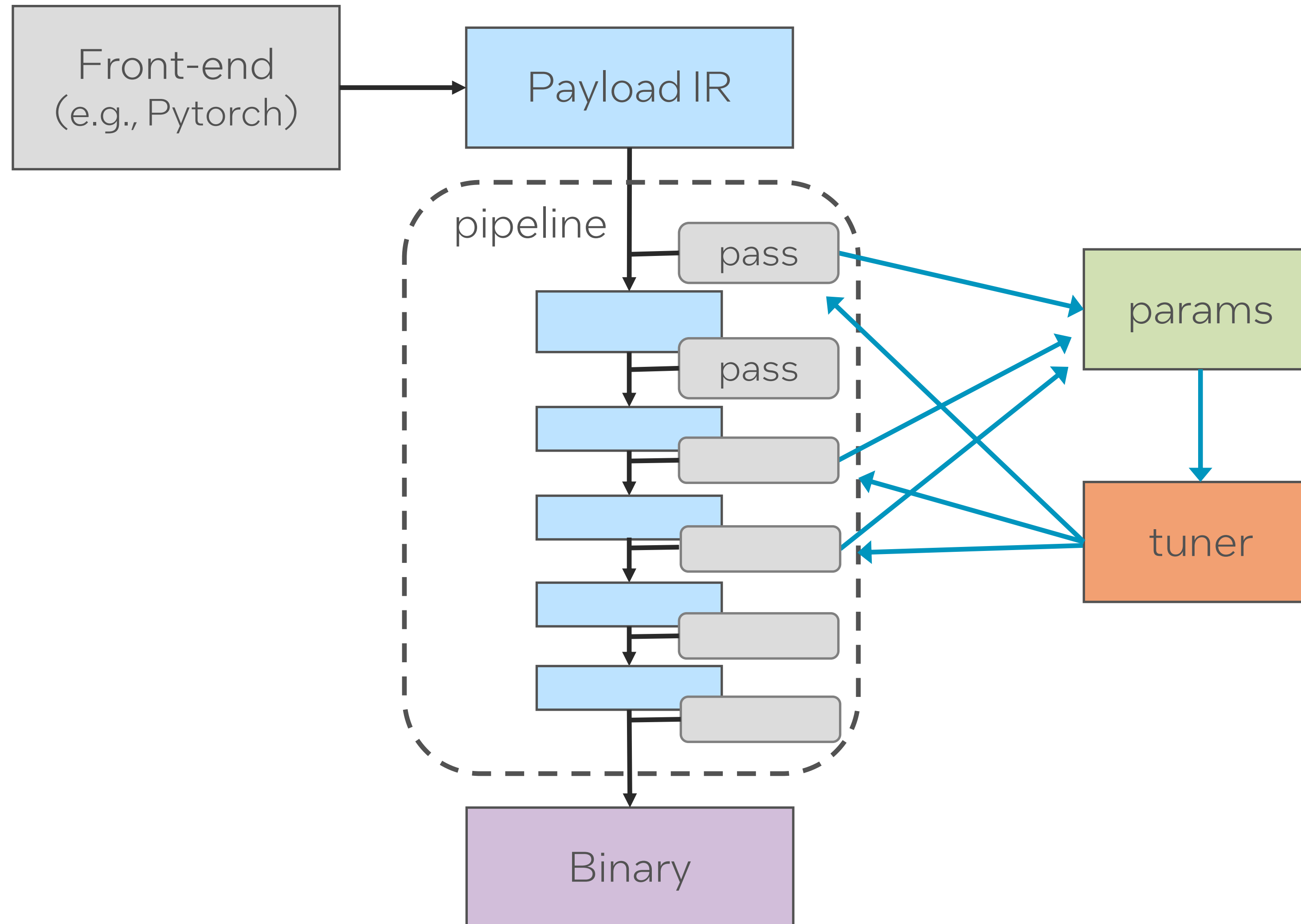
Auto-tuning MLIR schedules for Intel GPUs

Tuomas Karna (Intel), Rolf Morel (Intel)

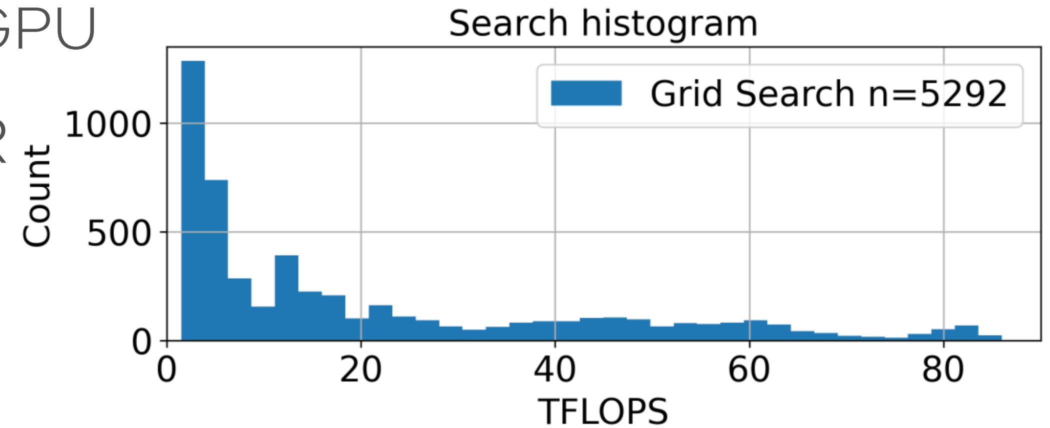
Euro LLVM Developers' Meeting, 2026
MLIR Workshop

Kernel autotuning in MLIR



Example: optimizing a GEMM kernel for Intel GPU

4k GEMM (f16, f16, f32) on Intel Arc B580 GPU

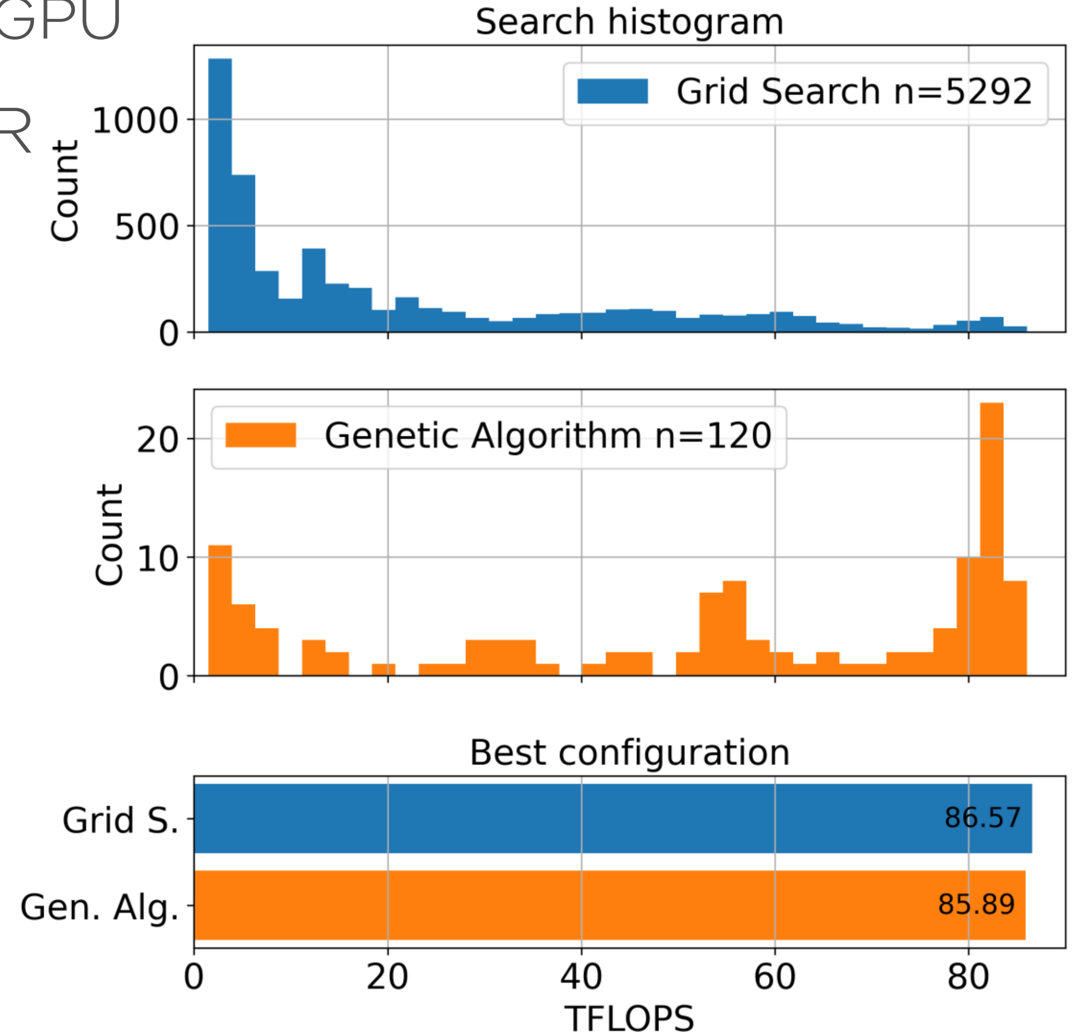


- End-to-end lowering with upstream MLIR
- 13 scalar parameters with constraints
  - Work/subgroup tile size, prefetch size, ...
  - Choices: wg_tile_m in [32, 64, 128]
  - M % wg_tile_m == 0, ...
- Complexity
  - **~2.7e6** possible combinations
  - Applying constraints: **5292** combinations
- Exhaustive grid search using MLIR
  - ~3 h 40 min run time

Example: optimizing a GEMM kernel for Intel GPU

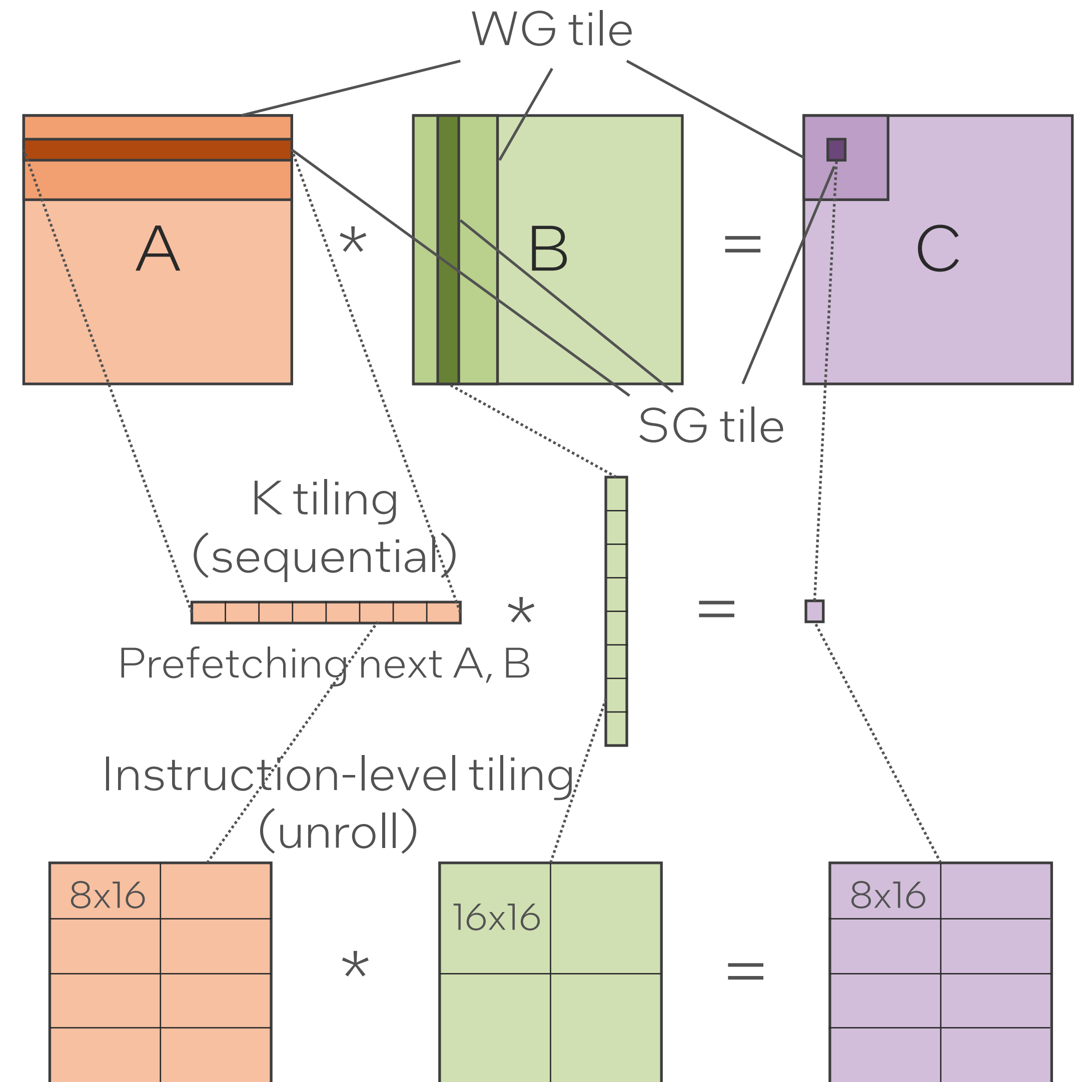
4k GEMM (f16, f16, f32) on Intel Arc B580 GPU

- End-to-end lowering with upstream MLIR
- 13 scalar parameters with constraints
  - Work/subgroup tile size, prefetch size, ...
  - Choices: wg_tile_m in [32, 64, 128]
  - M % wg_tile_m == 0, ...
- Complexity
  - **~2.7e6** possible combinations
  - Applying constraints: **5292** combinations
- Exhaustive grid search using MLIR
  - ~3 h 40 min run time
- Adaptive sampling
  - **~120** kernel evaluations, ~3 min run time

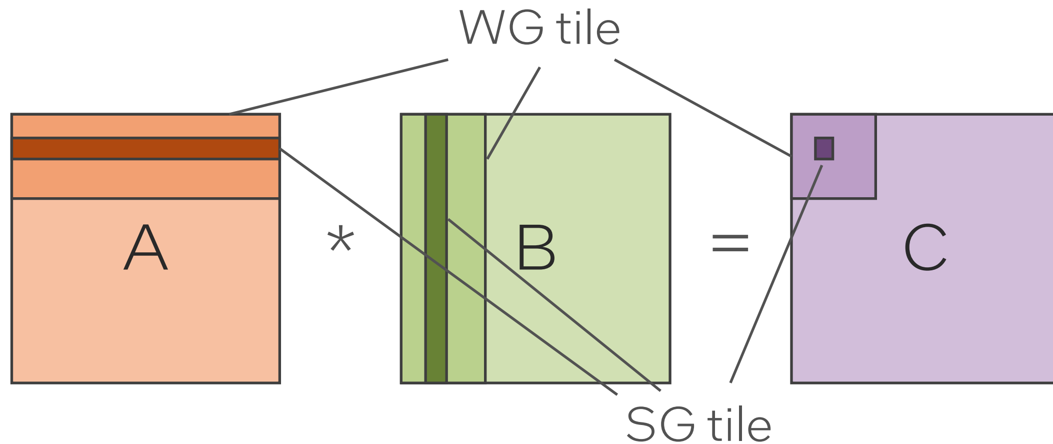


Matrix multiplication on Intel GPUs

- Intel® Xe Matrix Extensions (XMX)
  - DPAS op (Dot Product Accumulate Systolic)
  - Supported shapes
    - E.g., f16, f32: (8x16, 16x16) -> 8x16
- Hierarchical tiling of C matrix
  - Global, Workgroup, Subgroup (parallel)
- Tiling of reduction dimension K
  - Prefetching A and B in the reduction loop
  - Loading A and B using larger tile size
- Instruction-level tiling to DPAS shape



Matrix multiplication example: Knobs and constraints



- 13 scalar parameters
  - Work-/subgroup tile size (m,n) (4)
  - K tile size (1)
  - A,B prefetch tile size (m,n) (4)
  - A,B load tile size (m,n) (4)

Constraints:

- Perfectly nested tile sizes
  - Global, WG, SG, k-tile, DPAS
  - Modulo constraints:
    M % WG_m == 0,
    WG_m % SG_m == 0, ...
  - Xe DPAS op constraints
- Prefetching A and B
  - Tile size compatible with WG, k tiles
  - Xe prefetch instruction constraints
- Heuristics
  - Minimum number of threads > 16:
    (WG_m / SG_m) * (WG_n / SG_n) > 16
  - Constrained parameter *expression*

GPU kernel tuning as an optimization problem

- Integer optimization
  - Sparse integer space
  - E.g., tile size, candidates [16, 32, 64, ...]
- Highly constrained
  - Algorithmic constraints
    - Perfectly nested tiles; M % wg_m == 0, ...
  - Hardware constraints
    - E.g., mma instruction tile sizes
  - Joint constraints
    - Expressions of parameters + constraints
    - E.g., tile_a / tile_b < param_c
  - Heuristics
    - E.g., underutilization; nb_threads > 16

Need to be able to

- Express tunable parameters (knobs)
- Express constraints
  - Per parameter and jointly
- Check validity of a given combination
- Enumerate all valid combinations

MLIR's Transform meta-Dialect

transform IR describes transformations on *payload IR*

Payload IR:

```
!ty = tensor<128x128xf32>
func.func @f(%A: !ty, %B: !ty, %bias: !ty) -> !ty {
  %EMP = tensor.empty()
  %MM = linalg.matmul ins(%A, %B) outs(%EMP)
  %res = linalg.add ins(%MM, %bias) outs(%EMP)
  return %res
}
```

Transform Interpreter

Transform IR:

```
transform.sequence(%func_handle: !transform.op<"func.func">) {
  %mm_handle = transform.match "linalg.matmul" in %func_handle
  transform.tile_using_forall %mm_handle tile_sizes [16, 16]
}
```

Transformed payload IR:

```
func.func @f(%A: !ty, %B: !ty, %bias: !ty) -> !ty {
  %EMP = tensor.empty()
  %MM = scf.forall (%I,%J) in (8,8) outs(%emp = %EMP) {
    %i = affine.apply #map(%I)
    %j = affine.apply #map(%J)
    %a = tensor.extract_slice %A[%i, 0] [16, 128]
    %b = tensor.extract_slice %Bslice[0, %j] [128, 16]
    %o = tensor.extract_slice %out[%i, %j] [16, 16]
    %mm = linalg.matmul ins(%a, %b) outs(%o)
    scf.forall.in_parallel {
      tensor.par_insert_slice %mm into %emp[%i, %j]
    }
  }
  %res = linalg.add ins(%MM, %bias) outs(%EMP)
  return %res
}
```

The MLIR Transform Dialect: Your Compiler Is More Powerful Than You Think
(Martin Paul Lücke, Alex Zinenko, William S. Moses, Michel Steuwer, Albert Cohen)

Tuning schedules: selecting tile sizes

```
transform.sequence(%func) {
  ...
  %mm_handle = transform.match "linalg.matmul" in %func_handle
  transform.tile_using_forall %mm_handle tile_sizes [32, 64]
  ...
}
```

Schedules with choices: from knobs to constants

```
...
%mm = transform.match "linalg.matmul" in %func
%tile_m = transform.tune.knob<"tile_m"> options = [16, 32, 64]
%tile_n = transform.tune.knob<"tile_n"> options = [32, 48, 64, 80]
%tiled_mm, %loop = transform.tile_using_forall %mm tile_sizes [%tile_m, %tile_n]
...
```

walker extracts

```
tile_m ∈ { 16, 32, 64 }
tile_n ∈ { 32, 48, 64, 80 }
```

oracle picks

```
tile_m = 32
tile_n = 64
```

driver rewrites

```
...
%mm = transform.match "linalg.matmul" in %func
%tile_m = transform.param.constant 32
%tile_n = transform.param.constant 64
%tiled_mm, %loop = transform.tile_using_forall %mm tile_sizes [%tile_m, %tile_n]
...
```

Tuning schedules: joint constraints via SMT

Handling knob expressions

- Knob arithmetic and constraints expressed in SMT dialect
- E.g., a joint constraint tile_m * tile_n ≤ 32:

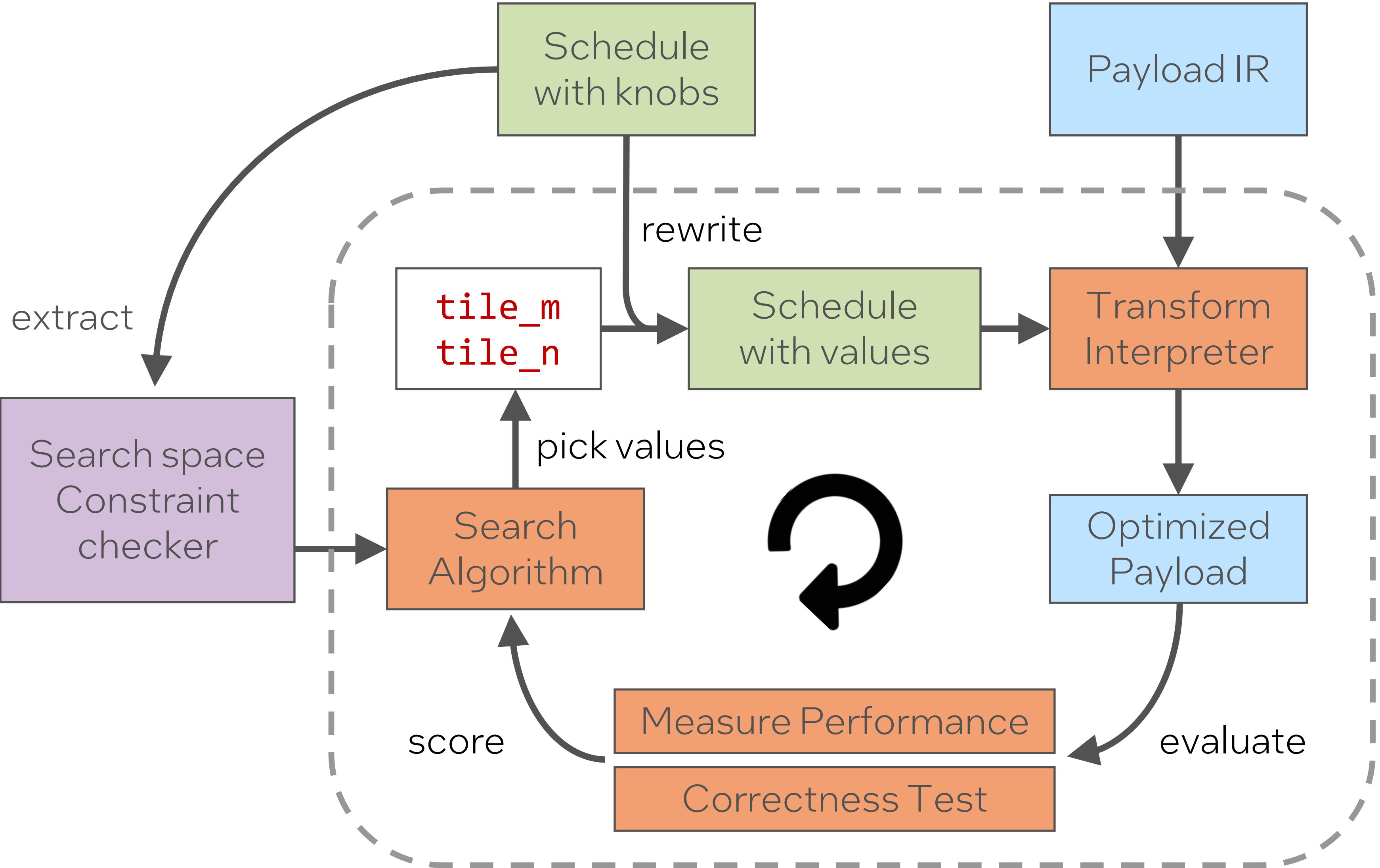
```
...
%tile_m = transform.tune.knob<"tile_m"> options = range<2,12> : !transform.param<i64>
%tile_n = transform.tune.knob<"tile_n"> options = range<2,12> : !transform.param<i64>
transform.smt.constrain_params(%tile_m, %tile_n) {
  ^bb(%tile_m_smt: !smt.int, %tile_n_smt: !smt.int):
  %m_times_n = smt.int.mul %tile_m_smt, %tile_n_smt
  %c32 = smt.int.constant 32
  %prod_le_32 = smt.int.cmp le %m_times_n, %c32
  smt.assert %prod_le_32
}
...
```

Params are converted to smt.int in the op body.

SMT arithmetic

SMT comparison + assert

Mechanized optimization of knobs



Systematic exploration: grid search and adaptive sampling

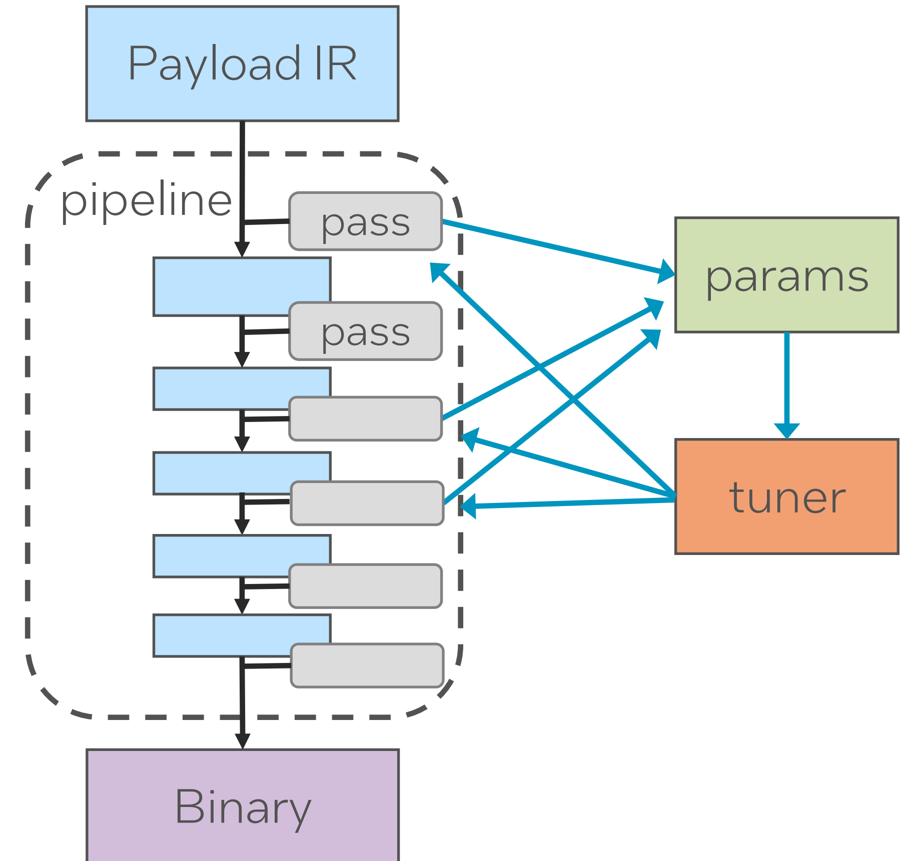
- *Exhaustive grid search*
  - Guaranteed to find optimal configuration
  - Can be prohibitively slow, even if sparse
- *Adaptive sampling*
  - Stochastic process to generate new "good" candidates
  - Constraint checker discards invalid ones
  - Faster, can miss global optimum

Genetic algorithm (proof-of-concept)

- Pool of n candidates
- Every generation
  - Generates n children based on fitness
  - Recombination and mutation
  - Next generation:
    take n best parent/child individuals
- Hyperparameters
  - Population size (=14)
  - Mutation rate (=0.001)
  - End criterion: n generations (=30)

Prerequisites for MLIR/tune dialect autotuning

- End-to-end lowering in MLIR
  - E.g., linalg to binary
- Including all necessary transformations to achieve best performance
- Identify and expose tunable parameters
  - Tile sizes, transformation choices, …
- Identify and implement constraints
- Also possible to use a pass pipeline



End-to-end lowering for Xe-GPU: linalg-to-binary

Upstream XeGPU pipeline:

XeGPU dialect

- Express GPU ops at workgroup, subgroup and instruction level abstractions.
- Passes: convert-vector-to-xegpu, xegpu-propagate-layout, xegpu-wg-to-sg-distribute, …

XeVM dialect

- Express ops close to ISA abstraction
- Passes: convert-xegpu-to-xevm

Finally: gpu-module-to-binary pass

New: XeGPU Transform ops

- Fill gaps in Linalg to XeGPU lowering
- Add layout annotations on XeGPU ops
  - Set subgroup and instruction tile sizes
- Insert prefetch ops
- Exposes necessary tunable knobs

```
%1 = transform.structured.match ops{["xegpu.store_nd"]} …
transform.xegpu.set_anchor_layout %1 sg_layout = [8, 8]
    sg_data = [32, 32] inst_data = [8, 16] : …
```

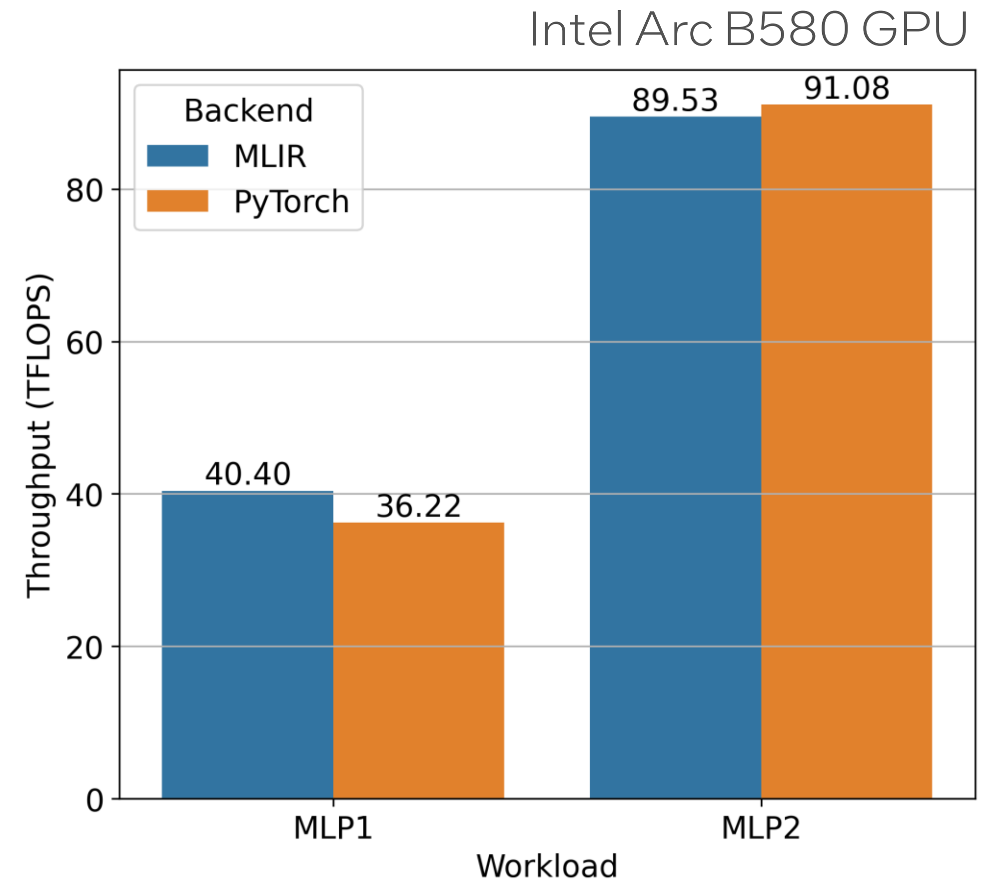
```
%1 = transform.xegpu.get_load_op %dpas_tile_a : …
%2 = transform.xegpu.insert_prefetch %1 nb_prefetch=1 : …
```

Autotuned MLP performance

- Comparing two MLP models
  - MLP1: Memory bound
  - MLP2: Compute bound
- Autotuned MLIR
  - All GEMMs tuned with genetic algorithm
- Reference: PyTorch
  - Intel GPU backend, OneDNN library

- Next steps:
  - Support more kernels
  - Softmax
  - …

Intel Arc B580 GPU



Name	Batch	Input	Output	Hidden layers
MLP1	128	16384	8192	16384, 16384
MLP2	4096	4096	4096	4096, 4096

All layers: f16 with f32 accumulator
ReLu activation on hidden layers

Conclusions

- GPU kernel tuning is a special integer optimization problem
  - Need to be able to express and check complex constraints
- Tune dialect
  - Tunable knobs expressed as transform ops, with list of choices
  - Constraints expressed with SMT-ops inside schedule
  - Automated construction of search space
- Grid search vs adaptive sampling
  - Adaptive sampling can be ~60x faster
- Intel GPU example
  - End-to-end lowering transform schedule in upstream MLIR
  - Includes tuning knobs and constraints
  - Autotuned MLP performance comparable to the state-of-the-art

Implemented in Lighthouse: https://github.com/llvm/lighthouse

Thank you!

Related talks:

Technical talks, Tuesday, April 14

- Renato Golin: *Lighthouse: infrastructure for end-to-end MLIR-compilers and testing.*
  1:15 PM - 1:45 PM
- Rolf Morel: *MLIR-iteration cycle goes BRRR: defining ops and rewrites in Python.*
  2:15 PM - 2:45 PM

Quick talk, Wednesday, April 15, 11:00 AM - 12:00 PM

- Adam Siemieniuk:
  Self-Contained, Target-Specific GEMM Code Generation in MLIR.

intel®

Disclaimer

- Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Learn more at intel.com, or from the OEM or retailer.
- No computer system can be absolutely secure.
- Intel processors of the same SKU may vary in frequency or power as a result of natural variability in the production process.
- Tests document performance of components on a particular test, in specific systems. Differences in hardware, software, or configuration will affect actual performance. Consult other sources of information to evaluate performance as you consider your purchase. For more complete information about performance and benchmark results, visit http://www.intel.com/performance.
- Some results have been estimated or simulated using internal Intel analysis or architecture simulation or modeling and provided to you for informational purposes.
- No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.
- Intel does not control or audit third-party benchmark data or the web sites referenced in this document. You should visit the referenced web site and confirm whether referenced data are accurate.
- Intel, the Intel logo and others are trademarks of Intel Corporation in the U.S. and/or other countries. *Other names and brands may be claimed as the property of others. © 2022 Intel Corporation.

intel®